Protecting COTS Binaries from Disclosure-guided Code Reuse Attacks*

Mingwei Zhang
Intel Labs
Hillsboro, OR, USA
mingwei.zhang@intel.com

Michalis Polychronakis
Stony Brook University
Stony Brook, NY, USA
mikepo@cs.stonybrook.edu

R. Sekar
Stony Brook University
Stony Brook, NY, USA
sekar@cs.stonybrook.edu

ABSTRACT

Code diversification, combined with execute-only memory, provides an effective defense against just-in-time code reuse attacks. However, existing techniques for combining code diversification and hardware-assisted memory protections typically require compiler support, as well as the deployment or modification of a hypervisor. These requirements often cannot be met, either because source code is not available, or because the required hardware features may not be available on the target system. In this paper we present SECRET, a software hardening technique tailored to legacy and closed-source software that provides equivalent protection to execute-only memory *without relying on hardware features or recompilation.* This is achieved using two novel techniques, *code space isolation* and *code pointer remapping*, which prevent read accesses to the executable memory of the protected code. Furthermore, SECRET thwarts code pointer harvesting attacks on ELF files by remapping existing code pointers to use random values. SECRET has been implemented on 32-bit Linux systems. Our evaluation shows that it introduces just 2% additional runtime overhead on top of a state-of-the-art CFI implementation, bringing the total average overhead to about 16%. In addition, it achieves better protection coverage compared to compiler-based techniques, as it can handle low-level machine code such as inline assembly or extra code introduced by the linker and loader.

1 INTRODUCTION

The deployment of non-executable memory protections in operating systems prompted a shift of attacks from code injection to code reuse, and in particular, return-oriented programming (ROP) [28, 38, 61]. After hijacking control flow, ROP attacks divert execution to code snippets ("gadgets") that already exist in the vulnerable process. One of the key requirements for ROP attacks is the knowledge of the memory locations of gadgets. Recent research [12, 64] has demonstrated that this requirement can be eliminated by exploiting a memory leakage vulnerability to harvest code pointers and disclose code memory on-the-fly. Armed with this knowledge, gadget chains can be constructed dynamically by malicious script code at the time of exploitation.

In the face of such "just-in-time" ROP (JIT-ROP) attacks, traditional code randomization defenses [10, 27, 35, 49, 72] do not offer any meaningful defense. As a response, recent proposals [6, 13, 17, 20, 32, 51, 66, 74] introduce a new security primitive that enforces diversified code pages to be executable but not readable. Such an execute-only policy can be implemented using page table manipulation [6], split TLBs [32], hardware virtualization extensions [20, 66, 74], or a form of software-fault isolation [13, 51].

A common characteristic of many of these approaches [13, 20, 32, 51] is that they rely on the recompilation of the target application, which is inconvenient at best, and impossible at worst (for code available only in binary form). As a result, they cannot be applied if source code is unavailable. In fact, even open-source software is typically distributed in a binary form, e.g., through package management systems such as `apt`. It is inconvenient for users to have to obtain all the necessary source code and recompile packages. Moreover, source-code based approaches are incomplete in that they do not protect low-level code written using inline assembly, or binary code that is automatically added by compilers and linkers. In contrast, techniques operating at the binary level can work seamlessly with the prevalent model of binary distributions, while protecting all code (including low-level code) that can potentially be used by an attacker.

A second limiting factor of many execute-only protections [17, 20, 32, 66, 74] is that they rely on hardware features that may not always be available on a given system. For instance, HideM [32] requires split TLB support, and is thus not applicable on current systems that use a shared code and data TLB. More recent proposals [20, 66, 74], on the other hand, rely on the extended page table (EPT) feature introduced in Intel VT-x, which allows setting code pages to be executable but not readable. Enabling this primitive requires support by both a thin hypervisor as well as the OS kernel. For already virtualized systems (e.g., cloud or enterprise environments) this will entail either nested virtualization, which may incur a significant runtime overhead [8] without architecture support, or the incorporation into existing hypervisors, affecting their performance and increasing the trusted computing base. For end-user or legacy systems, besides the fact that the deployment of new hypervisors and modified OS kernels is challenging, the required hardware support might not be available at all. For instance, NORAX [17] is applicable only on the AArch64 platform.

ACSAC 2017, December 4–8, 2017, San Juan, PR, USA

ACM ISBN 978-1-4503-5345-8/17/12…$15.00
https://doi.org/10.1145/3134600.3134634

*This work was supported in part by grants from ONR (N00014-15-1-2378 and N00014-17-1-2891) and NSF (CNS-1319137).

Overview of Approach and Summary of Contributions

In this paper we present SECRET, a protection against JIT-ROP attacks that is fully compatible with COTS binaries. SECRET enforces a policy equivalent to execute-only memory, with no reliance on any specific hardware features or any modifications to the virtualization setup. This achieved by enforcing a *"What You Target Is Not What You eXecute (WYTINWYX)"* property. This property is useful in thwarting attacks based on code-pointer harvesting: even if attackers find valid code pointers in memory, it is not possible to determine the code they point to. This not only applies to normal code pointers, but also to pointers that arise during exceptions. SECRET is based on static binary instrumentation, and is implemented over the PSI platform [78] (originated from BinCFI [79]).

The WYTINWYX property enforced by SECRET stems from two key techniques: *code space isolation (CSI)*, and *code pointer remapping (CPR)*. Code space isolation thwarts JIT-ROP attacks by (a) hiding the executing code in a large address space, turning it into *shadow code,* and (b) ensuring the absence of any pointers to it. Since static binary instrumentation systems such as BinCFI [79] and Reins [73] maintain the original code in addition to the instrumented version that is executed, JIT-ROP attacks could still work by reading the original code. We therefore present static analysis techniques to identify any embedded data and wipe out the rest of the original code. To break an attacker's ability to inject valid code pointers, code pointer remapping maps pointers to random values over a large range. This requires accurate identification of code pointers—a challenging problem for COTS binaries. SECRET leverages (when available) the DWARF and RTTI metadata typically contained in binary executables to accurately identify code pointers, and applies CPR to a wide range of pointers: return addresses (RAs), jump table pointers, and exported functions. Note that the vast majority of usable gadgets left unprotected by control flow integrity (CFI) approaches lacking a shadow stack correspond to RAs—there is very loose protection for such "call site gadgets" in coarse-grained CFI implementations, whereas SECRET constrains them effectively. Return address protections (e.g., shadow stacks) pose a significant compatibility challenge due to non-standard use cases [23]. CPR exploits the capabilities of address translation in a novel way to sidestep these challenges.

Completeness and *ease of deployment* are two key benefits of the proposed technique. By working directly at the binary level, SECRET achieves complete program instrumentation even for stripped executables and shared libraries, without the need for any recompilation which would complicate deployment. SECRET is applied to all code within a process, including low-level modules such as the dynamic loader (`ld.so`), system libraries (e.g., `libc.so`), and vDSO. In contrast, existing compiler-based execute-only memory protections leave out a significant amount of low-level code, such as hand-coded assembly, or code automatically added by linkers and loaders. Indicatively, `glibc` contains 56 KLoC of assembly code, *excluding* inline assembly. Additionally, compiler-based solutions cannot protect third-party libraries compiled using a different compiler. Systems like Readactor [20] also face compatibility problems with signals and C++ exceptions due to potential leakage of code pointers when they are stored in readable memory. By ensuring that no original code pointer points to shadow code, SECRET ensures that pointers cannot be leaked in such cases.

In summary, our work makes the following main contributions:

- We present two complementary static analysis techniques for the separation and protection of code against JIT-ROP attacks that rely on direct or indirect code disclosure. Code space isolation prevents direct code disclosure by moving code sections at random locations determined at load-time. Code pointer remapping thwarts indirect code disclosure through harvesting pointers from memory by replacing code pointers with randomized values scattered across a large address space.
- We have designed and implemented SECRET, a static binary instrumentation tool built on top of the PSI platform [78] that relies on CSI and CPR to protect COTS binaries on Linux.
- We experimentally evaluate SECRET and demonstrate its practicality. Our results show that SECRET protects all code, including low-level code that is available only in assembly or machine code format, while introducing a modest 2% additional runtime overhead over the base cost of PSI [78]. The total overhead, including CFI enforcement, is about 16%.

2 BACKGROUND

Static binary instrumentation (SBI) techniques instrument whole binaries prior to execution, while dynamic binary instrumentation (DBI) techniques perform instrumentation at runtime. DBI systems have tended to be more robust and provide better compatibility for complex code, but suffered from high performance overheads. SBI techniques significantly reduce these overheads, but have tended to be less robust on complex and/or low-level code. The primary goal of our earlier BinCFI [79] and PSI [2, 78] works was to address these robustness issues for large and complex binaries. In particular, we will summarize two features we use in this regard: (i) the use of two code versions, and (ii) address translation. This will be followed by a discussion of BinCFI's limitations against disclosure-guided code reuse attacks, which motivate the techniques developed in this paper.

Two code versions. Since data may be interspersed with code, it is not safe to overwrite original code, as this may result in overwriting of embedded data as well. Hence, many SBI systems leave the original code in place, and create a second instrumented copy that gets executed. The original code version is made non-executable, while the second (instrumented) code version is executable.

Since code pointer values may be stored anywhere in the data or code segments, it is not feasible to identify all such pointers with 100% accuracy. For this reason, BinCFI does not attempt to identify or modify these pointers, so all code pointers will continue to point to addresses within the *original* code. This approach, used previously in DBI systems, makes instrumentation transparent, and hence provides better backward compatibility. It works with applications that may use code pointer values for purposes such as C++ exception handling, computing the location of static variables, or to read their own code. Such code will examine the original code version, and hence avoid any confusion that may result from instructions introduced during instrumentation.

Address translation. As described above, code pointers continue to target the original code. This means that indirect control transfers

need special treatment, or else they will jump to the original (now non-executable) code. To avoid this, BinCFI uses address translation, a technique originally developed in DBI systems. In particular, code pointer values are translated just before their use in indirect control transfers, so that they will now point to the corresponding locations within the instrumented code version. This process, called *address translation,* relies on a hash table lookup at runtime.

BinCFI uses address translation not only to fix-up code addresses, but also to enforce CFI. In particular, control transfer instructions are grouped into classes such that all instructions in a class share the same set of valid targets. A separate address translation table is used for each class, and this table limits translations to only those targets that are valid for the class.

For modularity, each address translation table is divided into a *global translation table (GTT),* and a per-module (i.e., per binary file) *local translation table (LTT).* The GTT is populated by a modified loader, and it maps the most significant bits of an original code address to the corresponding module. The LTT of the module is then used to obtain the target address within that module.

BinCFI limitations against code disclosure-based attacks. Although BinCFI employs coarse-grained CFI that limits the available ROP gadgets, previous research [25, 33] has shown that a sufficient number of usable gadgets remain, and these can be used to achieve arbitrary code execution. Moreover, usable gadgets remain unchanged between the original and the instrumented versions of the code. Indeed, by reading the contents of GTT and LTT, an attacker can access the instrumented code version as well. As a result, even an attacker that doesn't know the original code can perform a JIT-ROP attack by reading the uninstrumented code version. The techniques described in this paper are hence necessary to thwart such disclosure-guided code reuse attacks.

One way to improve BinCFI is to remove gadgets available to attackers. However, this is not feasible since those are valid indirect targets that may be used by legitimate control flows. While fine-grained CFI approaches would reduce the average number of such gadgets available in each context, attackers may still find the "right" context where there are sufficient gadgets available, since they could read code. In fact, it is very likely that such code locations exist (e.g., code dispatchers) despite the use of fine-grained CFI. We therefore develop an alternative approach in this paper that relies on *hiding code* and *hiding code pointers.* This is a two-step approach: we hide the real executable code and remove the original code away. We then randomize code pointer values.

Benefits and challenges of our approach. An obvious benefit of hiding code is that it prevents gadget discovery by scanning code. In addition, by randomizing stored code pointer values, we break the attackers ability to reason about relative distances between pointers. For instance, they cannot read a return address from the stack, and then use it to target a gadget that occurs at a specific offset preceding (or following) it. Note that the use of randomized code pointers entails no new overhead: address translation needs to be performed any way, and it takes no extra effort to translate a randomized pointer, as compared to the original code pointer.

While there are many benefits to code hiding and randomizing stored code pointers, these techniques pose several new challenges as well. Hiding requires removal of the original code. This requires more accurate static analysis than the techniques used in BinCFI. Otherwise, any removal of embedded data would cause the program to crash, or function incorrectly. Code hiding also requires a dynamic code relocation, or else attackers will be able to identify the location of new code by simply adding a fixed offset to the base of the original (uninstrumented) code location. This too is unavailable in BinCFI, since the instrumented code is always appended just behind original binary.

Randomizing code pointer values is even more challenging, since it requires static identification and modification of code pointers. Static code pointer identification is known to be a very difficult problem on stripped binaries. Nevertheless, we have been able to develop techniques that can identify and randomize the vast majority of such pointers. We describe our approach in more detail in the following section.

3 SYSTEM DESIGN

Just-in-time ROP attacks rely on reading the code memory of an executing process to assemble gadgets on the fly. Such attacks may be used to bypass code diversification [24, 64], or to achieve reliability for frequently updated software [1, 5, 37]. SECRET thwarts such attacks using *code space isolation (CSI)* and *code pointer remapping (CPR)*, two novel techniques that realize an execute-but-no-read capability using only binary instrumentation (and no hardware or VMM support).

Recall that static binary instrumentation systems such as BinCFI [79] and Reins [73] maintain two copies of code: (i) the original code, which is readable but not executable, and (ii) the instrumented code, which is readable *and* executable. Hence, JIT-ROP attacks may operate by reading either of these copies. CSI precludes reads of original code by clearing it out. This is enabled by a static analysis approach we describe in Section 3.1.

We obviously cannot clear out the instrumented code, so we need alternative approaches to thwart attempts to read this code version. In the absence of any hardware or VMM features to prevent reads of code segments, there are three basic approaches an attacker could follow. The simplest approach is to find the base address of the code section(s) and scan them. CSI prevents this by locating instrumented code sections at random locations determined at load-time, so the attacker cannot predict these locations in advance. A second strategy available to the attacker is that of brute-force memory search. CSI renders brute-force scans impractical by distributing code over a very large address space. As a result, the probability of discovering a code page using a random probe is negligible.

These measures leave an attacker with only one option for examining code: reading data memory to discover code pointers, and following these pointers to inspect code. To thwart this class of attacks, we introduce a new technique called *code pointer remapping (CPR).* CPR replaces code pointers with randomized ("encrypted") values scattered across a large address space. *It is important to note that these code pointer values are unrelated to the actual code locations targeted by them.* This is made possible by the "magic" of address translation: at runtime, when an indirect control transfer instruction is executed, an "encrypted" code pointer value is translated into the correct location for the corresponding code.

Both CSI and CPR protections are applied on all modules and all low-level code and code pointers to prevent code reuse attacks.

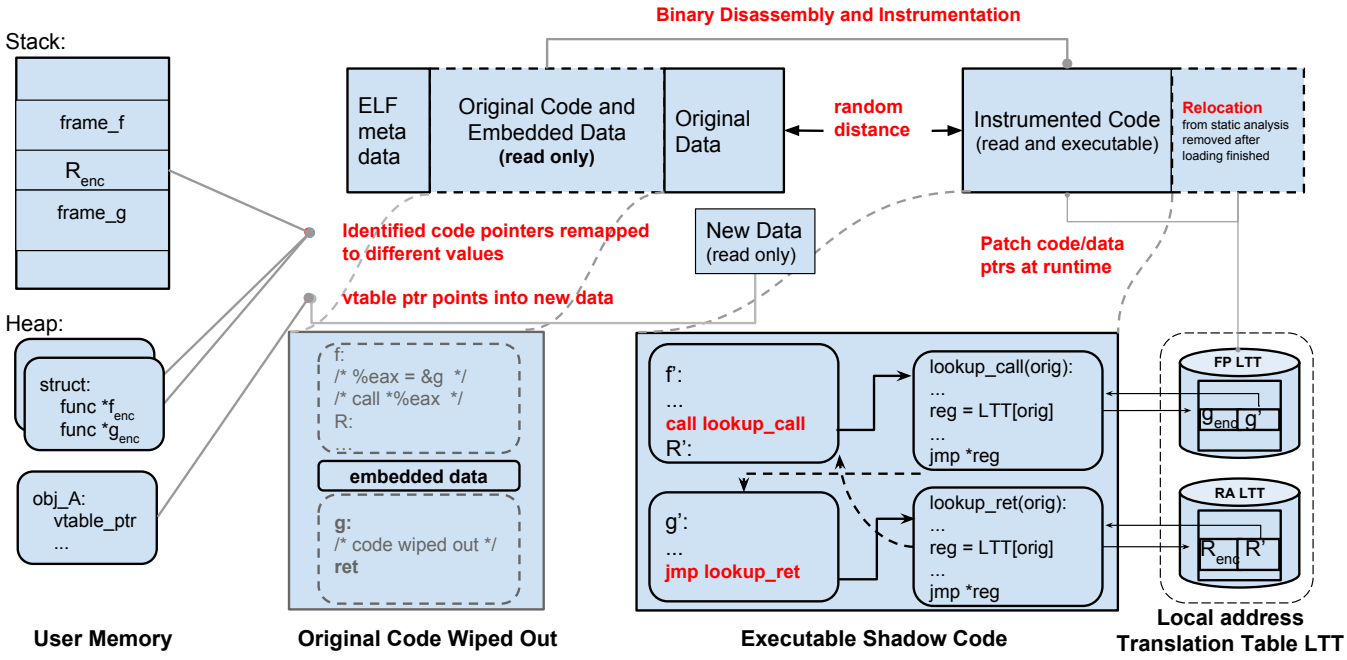

Fig. 1: Architecture of SECRET. Discovered code pointers are remapped to randomized memory regions, which is not related to either original code or shadow code. The actual instrumented code is hidden in a randomly allocated memory region with no data references to it. All code pointers are translated at runtime. During address translation, any accesses to the address translation table are performed through a private TLS to avoid memory leaks.

Compared to existing compiler-level techniques with the same goals [20], SECRET gains the following additional benefits due to its ability to operate on COTS binaries:

- *No changes to existing platforms:* Our technique does not require hardware support, OS support, modifications to compilation tool chains, or recompilation of existing programs. Besides the fact that source code is not always available, applying protections at the binary level is advantageous even for open-source applications, since it is compatible with the current deployment model of distributing identical binaries to every user.
- *Completeness of protection:* Defenses applied at the source code level may leave hand-written assembly code unprotected. More importantly, low-level code automatically inserted by tools such as linkers or compilers would also be unprotected. As discussed in previous research [19], many low-level code constructs such as context switches could be exploited by attackers to bypass existing defenses.
- *Transparent remapping and exception-based attacks:* By design, SECRET ensures that no original code pointer points to shadow code. Neither signals nor C++ exceptions leak code pointers. (Signal delivery is intercepted to modify the code address on the stack.) Both signals and C++ exceptions pose compatibility problems or leak pointers in the case of previous techniques such as Readactor [20].

3.1 Code Space Isolation (CSI)

Figure 1 illustrates our overall approach for protecting the original and the instrumented code. This figure illustrates both code space isolation, the topic of this section, and code-pointer remapping, a topic discussed in the next section.

CSI relies on static analysis to identify data embedded within the original code, and eliminates the rest of the code. CSI also incorporates techniques to decouple the locations of instrumented code from that of the original code. Decoupling is also applied to address translation tables (specifically, LTTs) that contain pointers to instrumented code. Finally, code that uses LTTs is implemented in such a manner that avoids storing instrumented code addresses in memory (other than the LTT itself). These measures turn the instrumented code into *shadow code* that is designed to be outside the attacker's reach.

3.1.1 Identifying embedded data

To properly identify the original code, we need to identify any data in the middle of code. To solve that problem, we have developed a static analysis pass to identify embedded data. The goal of this analysis is to be conservative: when in doubt, bytes should be marked as data and preserved, rather than being erased.

There are two types of embedded data: (a) data in the middle of functions, and (b) data between functions. The first type usually corresponds to jump table data, so we reuse existing techniques for jump table discovery to identify such data. For the second type of data, we leverage the information in two sections of COTS ELF binaries: `.eh_frame` and `.eh_frame_header`. These two sections are generated in the `DWARF` format used for C++ exception handling at runtime. Their purpose is to tell the C++ runtime how to unwind function frames. They include information such as function boundaries, the position of the saved frame pointer in a stack frame (or

the stack height, in case the frame pointer is not available), and the positions of callee-saved registers saved in the frame.

We have made an important observation about binaries generated from C programs, as well as several low-level binaries containing assembly code: *all such binaries contain* `.eh_frame` *and* `.eh_frame_header` *sections.* This is because C++ code may call non-C++ code and vice-versa. In order to properly handle exceptions, all function frames between the exception thrower and catcher must be available. In our experiments these two sections were available in all the COTS Linux binaries we have tested, including libraries with hand-written assembly code, e.g., `glibc`. Over 90% of the function boundaries were identifiable using this information. For the remaining cases, our current implementation falls back to the conservative binary analysis already incorporated into PSI. A better approach would be to use some of the recent techniques [54] for function boundary identification that achieve high accuracy.

3.1.2 Protecting shadow code

Our implementation platform places instrumented code at a memory location that follows the original code. Such an approach would make it easy for attackers to identify the location of shadow code. To prevent this, we redesigned the format of instrumented binaries in SECRET to decouple the locations of the original and instrumented code, turning instrumented code into shadow code.

To protect the shadow code in the 32-bit x86 architecture, we use segmentation to prevent its access from non-shadow code. This is achieved by isolating the shadow code as well as its LTT from any other readable user memory using a sandbox implemented by segmentation. The details of this technique are omitted since similar approaches have also been implemented in several previous works [4, 31, 39].

On architectures such as x86-64 where hardware segmentation enforcement is missing, software fault isolation (SFI) [71] can be used to protect the instrumented code, but the associated overheads can be significant. Moreover, instrumenting all memory accesses can be an engineering challenge due to the complexity of the x86 instruction set. We therefore opted for the alternative of base address randomization to protect the instrumented code. The large address space on x86-64 allows for sufficient entropy that makes guessing attacks very difficult if not impractical.

In our implementation, the shadow code of each module is located at a random distance from its original code. The random distance is determined at runtime by our modified loader, and can range over the entire address space available. The loading locations of different modules are determined independently.

3.2 Code Pointer Remapping (CPR)

Although shadow code has been isolated and hidden from the attacker, it still needs to be reachable. In particular, there will necessarily be pointers within data memory (stack, heap, or static areas) that can be used as control-flow targets. In the JIT-ROP threat model, it is impossible to prevent attackers from simply reusing such code pointers that they harvested by reading data memory. In other words, one cannot block attacks consisting *solely* of gadgets beginning at harvested code pointers. However, we want to prevent attackers from discovering additional usable gadgets from these harvested pointers. Specifically, CPR is aimed at blocking the following attack avenues:

(1) follow harvested code pointers to examine the shadow code and discover additional gadgets, or
(2) use prior knowledge of the victim program's code to compute the locations of additional usable gadgets, e.g., by adding an offset to a harvested code pointer value, or by repeatedly probing several nearby locations.

CPR achieves its goal by storing only *transformed code pointers* in memory. This transformation could be thought of as a cryptographic hash. Since a hash function cannot be inverted, it becomes infeasible to compute shadow code locations from the transformed code pointers stored in memory. This blocks the first attack avenue. For the second attack avenue, note that cryptographic hashes destroy correlations between their inputs, e.g., it is not possible to predict the hash of $x + 1$ given the hash of x. Thus, there is no way for an attacker to probe "nearby" gadgets.

For performance and other reasons, our implementation does not use a cryptographic hash. One limitation, dictated for compatibility with C++ exception handling, is that transformed code addresses of one function cannot be interspersed with that of another. However, we do transform the code locations within a function into an address space that is many orders of magnitude larger. Other than this need to avoid interspersing different functions, the transformed address space bears no relation to the actual locations where the target code is stored.

The CPR implementation consists of components that operate at instrumentation time and load time. At instrumentation time, CPR requires the identification of all code pointer constants in a binary, and their replacement with transformed values. Unfortunately, it is not always feasible on binaries to determine whether a constant represents a code pointer. As a result, a small fraction of code pointers are not transformed. However, these pointers will point to original code locations that have been cleared out by CSI, thus preventing them from being used to discover the location of shadow code.

At load-time, CPR requires changes to the system loader `ld.so`. At the time of loading a module, this modified loader reserves a range for transformed code addresses corresponding to this module. Code pointer values in the module are updated to use values over this address range. In order to speed up the loading process, we generate relocation information at code instrumentation time. This relocation information can be used by the loader to quickly fix up the code pointers within the module so that they use these transformed addresses.

In addition to replacing code references within the module, it is also necessary to set up the address translation tables so that they can map transformed code addresses into the corresponding locations where the shadow code is loaded. Specifically, the LTT needs to be updated so that it maps transformed code addresses into the corresponding locations within the shadow code.

Of the two components of the CPR implementation, the instrumentation time component is by far the most complex, and hence we describe it in more detail below.

3.2.1 Identifying code pointers

As discussed above, CPR requires the identification of code pointer constants in a binary, and replacing them with a transformed address. Clearly, such a transformation is safe only if we have very high confidence that we are dealing with a code pointer. However, it can be challenging to identify code pointer constants in COTS binaries with the requisite degree of confidence. We address this problem using a three-step approach:

- Develop static analysis techniques that are specialized for frequently used code pointer categories, e.g., return addresses and virtual functions.
- Develop a static analysis technique that can identify a subset of remaining function pointers with a high degree of confidence.
- Develop an approach for handling possible code pointers that are not detected in the previous two steps.

Sections 3.2.2 through 3.2.5 are devoted to the first step, while we detail our approach for the other two steps here. In particular, for the second step, our analysis identifies a constant as a code pointer if (1) it is an operand of an instruction or a constant inside data section and (2) it matches a function boundary address identified by DWARF section. Our experiments show that all code pointers inside SPEC benchmarks and binaries in coreutils could be correctly identified using this simple method. It is less successful on shared libraries.

For possible code pointers not recognized by the second step, we leave them as is, i.e., we do not remap them. So they will continue to point within the original code section. During address translation, these will be mapped into the corresponding locations within shadow code. As a result, compatibility will be preserved without leaking the location of shadow code. However, possible gadgets beginning at un-remapped pointers remain accessible using their original code addresses, and thus lose out the principal benefit of CPR. Fortunately, as we show in the evaluation, the vast majority of the pointers are remapped, so the number of gadgets accessible are rather small.

3.2.2 Remapping return addresses

Changing return addresses has two potential implications, since they may be used for purposes other than a return. Our experiments have shown that all such uses fall into one of the following cases on GNU/Linux:

- *C++ exception handling:* the return address is used to identify whether the caller of the current function has a handler for the current exception.
- *Caller checking:* the return address is used to determine the source of the call. Such checks occur in the dynamic loader.
- *PIC data access:* there are two cases: (a) jump tables, where the return address is popped off the stack and used to compute the base address of a jump table, and (b) static data accesses, where the return address is popped off the stack and an offset is added to find the base address for static data access.

For cases where return addresses are used for C++ exception handling, we update the corresponding `DWARF` metadata information. This is to ensure that the stack unwinding mechanism can work correctly with randomized return addresses. In particular, we update the `DWARF` information for each function by changing the function boundaries. The randomized function boundaries are currently equally distributed in the large random region for the whole module, but an alternative distribution that is proportional to the sizes of the functions could also be used. Since the C++ exception handling mechanism checks return addresses against their originators' function boundaries, to make such that checking works as intended, we must ensure that remapped return addresses still fall within their corresponding randomized function range.

The second case we have observed occurs in the dynamic loader, in which some internal functions check the location of callers. In particular, they require that callers only come from `libc.so` or `libpthread.so`. The check uses the loader's internal data structure `link_map`, which contains information about all modules. In order to cope with this, we change the `link_map` data structures so that base addresses correspond to the randomized address space. By doing so, the remapped return addresses can be correctly identified. In addition, to make sure all metadata can be accessed, we also adjust other related fields in `link_map`, such as offsets to metadata segments of the module. This is to ensure that our modifications are transparent to accesses of ELF metadata sections.

For the remaining cases, we rely on a static analysis pass to detect that the RA is being used as a data pointer, and avoid remapping it. The analysis determines that those addresses will not be used as part of actual return instructions (as they are popped off the stack), and avoids including them in the list of valid targets for return instructions.

3.2.3 Remapping C++ virtual functions

Since C++ programs on Linux follow the Itanium ABI, virtual function call sites and VTable assignments follow certain code signatures that can be captured statically [52]. We leverage metadata of COTS binaries, such as DWARF and runtime type identification (RTTI)[1], in combination with our static analysis, to identify VTables and virtual function pointers. Our current implementation supports all types of VTable recovery using RTTI, including those owned by multi-inherited classes.

As in previous work [63], we begin by scanning read-only data sections for the locations of the symbols `__class_typeinfo`, `__si_class_typeinfo`, and `__vmi_class_typeinfo` to recover all locations of `typeinfo` objects. In position-independent code (PIC), these symbol references are available as part of the dynamic relocation information, while in non-PIC, we identify the locations by searching the entire section. Using these `typeinfo` locations, we further scan the entire section for their references. Any location with a valid `typeinfo` address preceded by a zero is the base location of a VTable.

In case RTTI information is not available, we use static binary analysis to reliably recover VTable locations. In particular, we detect VTable assignment instructions using the following steps: *1) identifying constructor functions:* In C++ programs, creating an object usually requires calling a runtime function `new`, followed by its constructor function where VTables are assigned (we discuss exceptional cases in Section 6). We can easily identify all call sites

[1]DWARF sections are mandatory for C++ programs in Linux because of exception handling, while RTTI is optional but is by default turned on

of new (mangled names are _Znwj/_Znwm) and look into the next call instruction that takes the return value of new as the first argument. *2) identifying all VTable assignments:* We scan the code of callee functions using a simple data flow analysis. VTable assignments are identified by checking the following properties: a) a constant with a value pointing to read-only data is the source, and b) the target location is the head of the object (first argument of the constructor function). Due to multi-inheritance, multiple VTable assignments may exist in one constructor. To identify all other VTable assignments, we identify the co-appearance of their corresponding constructor functions. A constant assignment instruction is a VTable assignment only if it is preceded by a call instruction whose first argument matches the target address.

Once VTable base addresses and assignments are discovered, we proceed to detect VTable boundaries. Note that compilers such as gcc and llvm generate a VTable as a contiguous chunk of code pointers. However, in practice, a VTable may be contiguous with an adjacent VTable, rendering the boundary analysis incorrect. To deal with this challenge, we follow the approach of previous work [52] and conservatively scan each VTable linearly until we reach a non-code pointer such as zero. Note that, in the code of many libraries, several VTables may contain only zeros in the entire region. This happens usually because the class of the VTable is exported and all virtual functions can only be decided at runtime. We deal with this issue by following the dynamic relocation table present in COTS binaries.

Once all VTable assignment instructions and VTable boundaries have been determined, we relocate each VTable to a new data section and modify the corresponding vtable assignment instructions. This process eliminates any prior knowledge of an attacker about the original binary.

3.2.4 Exported functions and related code pointers

Remapping exported functions is necessary because these pointers will be propagated by the dynamic loader into the Global Offset Tables (GOT) of dependent modules. Attackers may use this information to infer other module base addresses. These pointers are remapped by updating the dynamic symbol table in each ELF image at runtime. Other than exported functions, there are several sections, such as .init_array, .fini_array, .ctors, and .dtors, which are known to contain code pointers.

3.2.5 Protecting jump table pointers

Code pointers in jump tables may point to useful gadgets. Because jump table pointers are computed in a register and used immediately afterwards, we rely on a simpler strategy that eliminates them from program memory. This is achieved by transforming code pointers used in jump tables and putting them into a new table along with the instrumented code (this means that the jump table is hidden from the program in the same manner as the instrumented code). In addition, we change indirect jumps that look up old jump tables to ensure that they check the corresponding new tables in the instrumented code. By doing so, jump tables will enjoy better performance since no translation is needed. In addition, jump table targets are eliminated from the address translation table to improve the strength of the CFI policy.

Table 2: Code pointer remapping coverage. Column 2 shows the total number of code pointers, columns 3–6 the number of remapped code pointers, and the last column the percentage of code pointers that have been protected.

Name	Total	Return	Jump Table	Exception Handlers	Exported Funcs.	% Remapped
400.perlbench	17548	14101	1542	0	3	89%
401.bzip2	512	365	48	0	2	80%
403.gcc	58264	47847	6496	0	7	93%
429.mcf	146	94	0	0	2	66%
445.gobmk	12220	9245	266	0	3	78%
456.hmmer	4426	3624	223	0	2	87%
458.sjeng	1436	1124	140	0	3	88%
462.libquantum	585	448	1	0	2	77%
464.h264ref	3738	3059	89	0	3	84%
471.omnetpp	21116	16700	956	3708	23	94%
473.astar	535	411	3	0	2	78%
433.milc	1881	1561	38	0	2	85%
435.gromacs	8491	6936	321	0	19	86%
437.leslie3d	693	631	0	0	2	91%
444.namd	1343	1157	8	16	3	87%
447.dealII	48927	37679	2801	6632	10	89%
450.soplex	6668	5237	570	493	5	91%
453.povray	13887	10559	1702	103	26	89%
454.calculix	19318	17699	206	0	2	92%
470.lbm	126	79	0	0	2	64%
482.sphinx3	2930	2530	5	0	2	87%
libc.so.6	26719	12117	12432	0	2163	98%
Total	251509	193203	27847	10952	2288	90.7%

4 EVALUATION

Since the base PSI platform works only on x86-32 Linux, we also implemented SECRET on the same platform. We implemented both segmentation-based and randomization-based protection for the shadow code and all related data structures. Our evaluation was carried out using SPEC benchmarks and a few real-world applications, including GUI applications such as Open Office. Unless stated otherwise, experiments were performed on a 32-bit Ubuntu system equipped with a Core i5 CPU and 4 GB RAM.

4.1 Effectiveness Evaluation

4.1.1 Code Pointer Remapping

We have evaluated the effectiveness of code pointer remapping on the SPEC benchmark programs as well as libc. Table 2 shows the fraction of code pointers that were remapped, for different types of pointers. Our analysis illustrates that the majority (90%) of code pointers have been handled, including all return addresses, all C++ virtual function pointers, all jump table pointers, and a subset of function pointers, as described in Section 3.2.

As described in Section 3.2.1, we use a conservative approach for identifying code pointers for remapping. In particular, for constants that appear to be code pointers but cannot be confirmed using the first two steps described in Section 3.2.1, we leave them as is, and do not remap them. This is the reason why approximately 10% of the code pointers is left unremapped in the results shown in Table 2. However, since about 90% of the code pointes have been remapped, the ability of attackers to construct successful ROP payloads is significantly constrained. To support this claim, we performed an an experimental evaluation using ROP payload generation toolkits. We used ROPGadget [56] and Q [58] in this evaluation. Neither tool was able to generate meaningful attack payloads using only the gadgets beginning at unremapped code pointers.

Table 3: Number of remapped vtables and virtual functions.

Name	# of Virtual Tables	# of Virtual Functions
omnetpp	120	2572
soplex	29	790
dealII	727	2454
povray	28	86
namd	4	8
astar	1	3

Another interesting point is the experiment on libc.so (last line in the table), as libc.so is a low level binary file that is compiled from C code mixed with inline assembly code. Our system has remapped 98% of code pointers. In addition, our experiment shows that 56K lines of assembly code have been compiled into libc.so. In this fraction of glibc code, there are 675 code pointers generated by call instructions. All these code pointers are protected by code pointer remapping.

Table 3 shows the results of remapping vtable and virtual function pointers. In particular, we have remapped *all vtable and virtual function pointers in the C++ benchmark programs used in our evaluation,* as well as their dependent libraries. As a result, advanced code reuse attacks such as COOP [57] are defeated due to the randomization of virtual table addresses, as the remapped virtual tables are located in a new data segment independent from the original code. Consequently, it becomes challenging to retrieve further information even if some of vtable pointers are leaked.

As previously discussed, all code pointers in a module are randomly remapped to the shadow address space. To properly support C++ exception handling, code pointer remapping is performed according to the DWARF frame description entry (FDE), i.e., code pointers within a piece of code covered by a single FDE (usually a function) should be remapped into a contiguous region in the randomized address space. As a result, if one of the return addresses within an FDE is leaked, then the entropy of other return addresses within the same FDE goes down to the size of that FDE in the randomized address space. Our experiments show that the entropy of these return addresses is 20 bits on average, while the entropy of other remapped pointers can be 32 bits.

4.1.2 Identification of embedded data

We evaluated the ability of our static analysis for discovering embedded data within code sections. As described earlier, the `.eh_frame` section provides information on how to unwind stack frames. The covered region consists of a list of debugging units, each of which usually corresponds to a function or a code snippet. The frame description entry (FDE) structure includes the range of the code in each case. Table 4 shows the exception handling information coverage for a set of SPEC binaries and Linux libraries. We summed up the ranges of all entries and show in the second column how much code was covered by the `DWARF` information. On average, 97.17% of the code is covered, which means that accurate function boundary information is available for almost all of the functions of these binaries.

With those boundaries as starting points, SECRET's static analysis pass can follow control flows within the already known regions and discover any missing code, as well as data in between and in the middle of functions. After analyzing 491 ELF system binaries in Ubuntu 14.04, we have found a few cases of data embedded in code. However, in most of these cases, the gap region indicated in the `.eh_frame` section was simply the padding data in between function or section boundaries. There were only a few cases in which data was embedded in the code as part of jump tables. Table 5 provides details about these cases. We found 40 locations totaling 390 bytes of data in `libc.so.6`, all of them used as padding. Since the value of the padding is zero, they can cause disassembly errors if not handled properly. In `libffi.so.6` on x86, we found a jump table in the middle of code inside the `ffi_call_SYSV` function. The same library on x86-64 has two jump tables identified by our algorithm. Finally, `libcrypto.so.1.0.0` contains 16 data regions (corresponding to approximately 20KB) in the middle of code in both 32-bit and 64-bit versions. All these data regions are located after function returns.

Table 4: Coverage of exception handling (DWARF) information.

Name	`.eh_frame` Coverage
spec2006	97.54%
libc.so.6	97.87%
libm.so.6	96.16%
libgfortran.so.3	98.58%
libquadmath.so.0	99.63%
libstdc++.so.6	95.44%
libcrypto.so.1.0.0	87.23%
Average	97.17%

Table 5: Embedded data regions identified by static analysis.

Name	Invalid Regions	Valid Regions	Reason
libc.so.6	40	0	Alignment padding
libffi.so.6	0	1	ffi_call_SYSV
libcrypto.so.1.0.0	0	16	Lookup table for crypto algorithms

4.1.3 Randomization Entropy

To ensure that our shadow code leverages the full entropy of the address space, we implemented our own code loading primitive as part of our modified loader to hide the location of shadow code. Although SECRET is currently limited to 32-bit systems, we wanted to evaluate achievable entropy on 64-bit systems. We hence ported our modified loader in the x86_64 glibc 2.19 running in Ubuntu 14.04. We then performed an experiment using Chrome 43.02 by forcing our modified loader to load the browser code as well as its dependent libraries. In this experiment, we used instrumented code of the same size as the code in the original binary, i.e., when a module is loaded, our loader immediately loads a corresponding code piece whose size is the same as the text segment of the module. In our experiment, all 24 processes of Chrome were tested.

Our experiments illustrate that the size of the instrumented code used by chrome is 514 MB. The instrumented code pages allocated are scattered in the whole user address space. The address range is different on each process, but the overall range of the address space that instrumented code occupies across different processes is between 953 and 1021 TB. Since instrumented code is not targeted

Table 6: Low-level indirect control transfer instructions protected by SECRET.

Name	Return	Indirect Jump	Indirect Call	Syscall
glibc	603	146	675	2
vDSO	1	0	0	1

by any code pointers, the probability of a memory leak is calculated as the size of the code divided by the address space used, which is about 5×10^{-7}.

We note that isolation based on information hiding may not be strong in some scenarios, especially if an attacker has the capability of probing information about the memory layout using timing channels [30], or other side channels such as the size of unallocated memory [47]. At the same time, it is clear that high entropy randomization substantially increases the attacker's work factor.

4.1.4 Low-level Protection Coverage

To evaluate the completeness of our approach, we have evaluated several low-level binaries that are used by most programs: ld.so, libc.so, libgcc_s.so and vDSO. The hand-written assembly code contained in these binaries exceeds 56K LoC.

Table 6 shows all the low-level indirect control transfer instructions that are protected by SECRET. In particular, we found 218 call instructions in native assembly code in glibc. The rest of low-level calls (457) were used for system calls (e.g., `call %gs:0x10`). In addition to low-level calls, we have found that three quarters of indirect jumps are low-level instructions. In particular, we found 256 indirect jumps written in assembly code, 230 of which are used as part of jump tables. Our experiments illustrate that there are 1545 code pointers used by these low-level jump tables and 457 return addresses could be generated by low level calls. All these low-level code pointers are fully protected by SECRET.

4.2 Runtime Performance

4.2.1 SPEC 2006 Benchmarks

We have evaluated SECRET's runtime overhead using the SPECINT 2006 benchmarks. Since code space isolation does not introduce any extra overhead, we include this feature on by default except in the baseline BinCFI system. We compare between three different modes: 1) `BinCFI`: baseline protection; 2) `SECRET.seg`: shadow code protected using memory segmentation; and 3) `SECRET.rand`: shadow code protected using base address randomization. In all cases, SECRET transforms the main executable and all six dependent libraries. The results for each benchmark are shown in Figure 7, while Table 8 shows the average overhead for SPECINT, as well as the total for all 21 SPEC CPU benchmarks.

In the `SECRET.seg` mode, the average overhead for SPECINT is 14.41% (the total SPEC CPU overhead is 15.64%). In this mode, both the instrumented code and its LTT are located outside of the memory sandbox. The overhead in this mode mostly comes from memory access through a segment register when performing address translation. In the `SECRET.rand` mode, the average runtime overhead for SPECINT is 13.54% (the total SPEC CPU overhead is 14.48%). Compared with `SECRET.rand`, there are two main differences in this mode: (a) the address translation trampolines perform

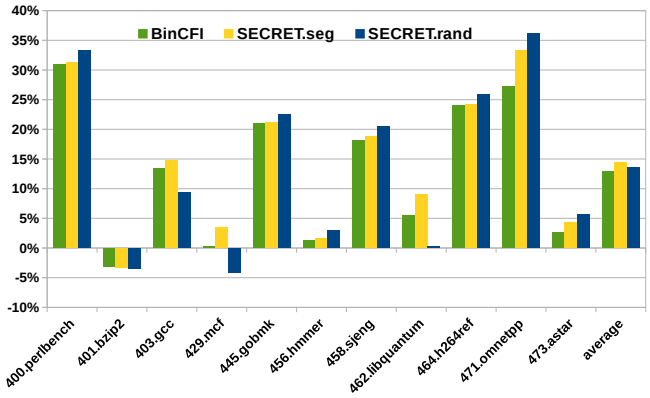

Fig. 7: Runtime overhead of SECRET on the SPECINT 2006 benchmarks.

Table 8: Summary of SPEC 2006 runtime overhead results.

Programs	BinCFI	SECRET.seg	SECRET.rand
SPECINT	12.84%	14.41%	13.54%
Total	14.20%	15.64%	14.48%

Table 9: Completion time (sec) for real-world programs.

Test Suite	Base	SECRET.rand	Description
python	4.709	5.022	Run bincfi script to transform /bin/ls
dd	99.46	99.6	Copy a 1GB file
md5sum	2.44	2.45	Checksum of file (1GB)
scp	2.78	2.96	Copy a 100MB file

two range checks (one for the original and one for the randomized code address space) instead of one, adding a bit more overhead, and (b) `SECRET.rand` does not require intensive memory access through our TLS. Our experiments show that the average overhead of `SECRET.rand` is slightly lower than `SECRET.seg`. Compared with the baseline system, the average runtime overhead added by `SECRET.seg` is less than 2%, and by `SECRET.rand` is less than 1%.

In our experiments, the average code size increase on SPEC programs was 3x, mostly due to LTTs, which consume a large amount of space. Code pointer remapping also contributes to the size increase since it duplicates many jump tables. Other than binary size, we also measure the physical memory overhead at runtime. In our experiment, for simplicity, we measured the peak usage of resident memory for each SPEC program. We observed that SECRET uses up to 4.3% extra memory over the original binary. This is reasonable despite the larger increase of binary size, since most of the SPEC benchmark programs allocate large chunks of data at runtime, and hence their memory use is determined more by data memory size rather than code size.

4.2.2 Commonly Used Applications

In addition to the SPEC benchmarks, we also evaluated SECRET with several real-world programs. As the SECRET.rand mode includes all features and supports the recent x86-64 architecture, we used this mode to compare with the performance of the original programs. The results of Table 9 show that SECRET is practical

Table 10: *Startup* overhead for launching GUI programs.

Name	Base (sec)	BinCFI	SECRET.rand
vim	0.6	60%	67%
lynx	0.02	100%	100%
evince	0.34	135%	168%
gcalctool	0.62	110%	161%
gedit	0.6	120%	165%
LibreOffice	1.4	51%	200%

for real-world usage. This experiment includes script interpreters (`python` and `perl`), disk I/O tools (`dd`), as well as network related tools (`scp`). In all experiments, the code of all main executables and libraries was transformed to shadow code.

We also evaluated further the startup overhead of protected programs, as this may affect user experience. SECRET has noticeable startup overhead due to its modified loader, which needs to perform the following actions on each module: (a) load the instrumented code as well as the LTT (b) initialize the corresponding entries in the GTT, for code pointer remapping to work properly, and (c) wipe out the original code. To better assess this overhead, we used a set of GUI applications, since they typically depend on many more libraries compared to the simple benchmark programs. As in the previous experiment, we use the SECRET.rand mode to compare with the original programs and the baseline system. Table 10 shows the startup overhead of several well-known Linux applications, including three GTK and two text user interface programs. The results show that SECRET's overhead is higher on GTK programs than programs using a textual interface. This is because GTK programs load many more libraries at program start up.

5 RELATED WORK

5.1 Control Flow Integrity

Control flow integrity (CFI) [3] provides a principled foundation for enforcing low-level security policies on binary code. The main idea of CFI is to mediate indirect control flow transfers and permit only allowed targets. CFI can be informally classified to coarse-grained [77, 79], fine-grained [45, 46, 50, 68], and context sensitive [44, 69], depending on the enforced policy. Although CFI enforcement makes code reuse exploits much harder, researchers have shown that they are still possible [16, 25, 33, 34]. These attacks exploit the fact that any static analysis used to infer intended control-flow must be approximate, and hence cannot prevent attacks that exclusively use gadgets that are determined to be legitimate by the analysis. This factor motivates the approach developed in this paper so that SECRET can provide stronger protection than what is achieved using the coarse-grained CFI provided by our platform PSI. Our approach complements CFI, including some of the recent advances [50, 52, 70], by making it much harder to discover gadgets (by hiding code), and to target them (by randomizing code pointer values).

Most of the CFI techniques referenced above are focused on *forward edges,* which include indirect calls and indirect jumps. Researchers have noted that protection of *backward edges,* i.e., returns, is even more critical. Indeed, ROP attacks repeatedly violate backward edge policies. Shadow stacks [18, 22, 26, 53] are a powerful mechanism for highly accurate enforcement of backward edge policies. However, shadow stacks experience compatibility problems in complex code due to non-standard use cases where return addresses are generated by non-call instructions. Rui et al [55] developed a static analysis to discover such non-standard cases, thus developing a robust shadow stack defense. SECRET provides fine-grained protection for backward edges by randomly remapping return addresses. Non-standard use cases don't pose a problem on our PSI platform: code addresses generated by any non-call instruction will be identified by PSI's static analysis as a possible code pointer analysis, and its use as a jump or return address will hence be permitted.

Approaches complementary to CFI have also been developed to defend against ROP. G-Free [48] implements an instruction transformation technique at the very last phase of compilation to eliminate unintended gadgets. However, intended gadgets (i.e., legitimate return targets) still pose a problem. Control-flow and code integrity (CFCI) [80] limits the use of intended gadgets so that they cannot be used to achieve the common attacker goal of loading injected code, e.g., by executing an mmap call to make data executable.

5.2 Code Randomization

PaX team introduced one of the earliest implementations of address space layout randomization (ASLR) [42] and non-executable memory pages [67]. ASLR is an important and defense that mitigates code injection as well as code reuse attacks. However, it is known that coarse-grained randomization, as used in PaX and other early ASLR implementations [9, 40, 76] and is in wide use today, has several weaknesses [29, 62]. Information leakage attacks are arguably the biggest threat to ASLR today. By disclosing the base address of a dynamically loaded module, the exploit code can dynamically adjust the gadget addresses used in the (pre-constructed) ROP payload, and effectively bypass ASLR [29].

To thwart this attack, fine-grained code diversification offers an additional layer of protection over ASLR, by randomizing not only the location but also the internal structure of code within a code section. Code diversification can be applied at varying granularities, e.g., at the function [10, 36, 43], basic block [72], or instruction granularity [27, 49].

Code randomization techniques that operate on source code are capable of fully randomizing code locations. In contrast, techniques that operate on COTS binaries can perform only a limited set of conservative randomizations, e.g., in-place randomization [49]. This is because of the previously identified difficulties in accurate identification of code pointers in binaries.

5.3 Code Disclosure Attacks and Countermeasures

Code randomization is challenged by advanced attacks that leverage memory disclosure vulnerabilities along with scripting capabilities to dynamically construct ROP payloads [12, 64]. Such "just-in-time ROP" (JIT-ROP) attacks [64] repeatedly use a memory disclosure capability to read executable memory and chain discovered gadgets to launch a ROP attack. BROP (blind ROP) [12] leverages a stack buffer overflow in forking servers to repeatedly overwrite the stack

until the `write` function is located, which then is used to leak executable process memory to the client. Under certain circumstances, even if a memory disclosure bug is not available, gadget locations can be inferred through side channels [60].

Recent research extends code randomization with dynamic re-randomization to thwart JIT-ROP attacks. Bigelow et al. [11] propose TASR, a re-randomization approach that randomizes the code upon each system call. Shuffler [75] provides a continuous code re-randomization capability. However, TASR requires source code, compiler, and kernel support, while Shuffler works on binaries but relies on the compiler to provide symbolic and relocation information. SECRET, in contrast, operates on stripped binaries without needing any such information.

Oxymoron [7] applies fine grained code randomization that is compatible with code sharing. However, page-level code randomization has been proven ineffective by Isomeron [24], which shows that even leaking one page of memory may still allow a successful ROP attack. Isomeron [24] thwarts JIT-ROP attacks by creating execution path diversity with multiple code versions. Although attackers can still read code, they do not know which version will be actually executed. Therefore, the possibility of successful gadget chain execution drops exponentially.

Another line of recent research efforts has focused on enforcing an execute-only memory policy to prevent JIT-ROP attacks from reading gadgets from memory [6, 13, 17, 20, 32, 51, 66, 74] This can be achieved using page table manipulation [6], split TLBs [32], hardware virtualization extensions [20, 21, 66, 74], or a form of software-fault isolation [13, 51]. For instance, Readactor [20] and Readactor++ [21] rely on the extended page table (EPT) feature of Intel processors. In addition, they protect all code pointers by forcing them to point to "proxy" pages that contain trampoline code stubs. By doing so, JIT-ROP attacks that harvest code pointers are defeated because leaked code pointers all point to non-readable "proxy" pages that leak no further information to attackers. SECRET provides a similar capability by using code pointer remapping without relying on recompilation or special hardware features.

LR2 [14] prevents code and code pointer disclosure similarly to our work. LR2 focuses on low-end ARM devices and confines memory reads on a certain memory range by masking load instructions. All control flows to a function are intercepted by trampolines which use direct jumps to relay control. LR2 operates at the source code level, while SECRET operates directly on stripped binaries.

5.4 Dynamic Binary Instrumentation

Dynamic binary instrumentation (DBI) systems [15, 41, 59] use a code cache to execute translated application code. Similar to shadow code, the code cache is isolated, and indirect control transfer targets are translated using an address mapping table. For performance reasons, the code cache is isolated. For performance reasons, code cache usually remains both writable and executable, which subjects it to code corruption attacks. In contrast, shadow code is never writable, so attackers cannot corrupt it. Recent research [65] has shown how to secure code cache using two processes, one for code generation and another for execution, but the technique has not yet been incorporated into the above DBI platforms.

An important difference between a code cache, as used in DBI systems, and shadow code is that the latter is self-contained and executes independently, while the former requires constant orchestration by the DBI runtime. This orchestration requires many data pointers in the code cache that point to critical data structures of the binary translator, leaking the locations of both.

6 LIMITATIONS

Use of RTTI. Given our focus on COTS binaries, our ability to identify code pointers is limited due to the lack of certain types of information. For instance, with the help of RTTI in C++ programs, we can reliably discover all virtual function information. When such information is not available, however, we can only use conservative static analysis, relying on the direct data flow between constructors and the `new` function. However, when an object is created on the stack, the `new` function is not called, since memory can be easily allocated on the stack. In such cases, SECRET cannot detect these virtual tables. Fortunately, the majority of binaries do contain RTTI, and most objects are allocated on the heap instead of the stack. For code pointers involving non-virtual functions, we are still working on further conservative techniques to improve upon their identification.

Applicability to x86-64 platforms. Our current prototype currently supports only 32-bit x86 platforms, given the fact that it is built on top of PSI, which only supports 32-bit systems. A few aspects of SECRET have been implemented and evaluated on x86_64, as described in Sections 3.1.2 and 4.1.3. A more complete implementation requires significant implementation effort, but is not conceptually more challenging.

DWARF information. One could argue that using DWARF information contradicts the claim of handling COTS binaries. The DWARF information used by SECRET is not the same as the optionally generated debugging information, which is also emitted in the DWARF format. The information we rely on is solely the exception handling information located in the `.eh_frame` and `.eh_frame_hdr` sections. This information is present even in *stripped* binaries, as it is critical for exception handling in C++ programs, and even in C programs, when stack unwinding involves stack frames of both C and C++ code.

7 CONCLUSION

Defending against advanced code reuse attacks that take advantage of memory disclosure vulnerabilities is becoming increasingly important. To that end, breaking the ability of attackers to read the executable memory segments of a process, or even to infer the location of potential gadgets, can be a significant roadblock. In this paper, we have achieved the above goal by designing and implementing SECRET, which introduces two novel code transformation techniques, *code space isolation* and *code pointer remapping*. The former prevents read accesses to the executable memory of the instrumented code (a protected version of an application's original code), while the latter decouples its required code pointers from that of the original code. Our experimental results demonstrate that SECRET can protect real-world COTS applications, while incurring only a modest performance overhead.

REFERENCES

[1] 2013. MWR Labs Pwn2Own 2013 Write-up - Webkit Exploit. (2013). http://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/.

[2] 2014. PSI Version 1.1. http://www.seclab.cs.sunysb.edu/seclab/psi/. (2014).

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *CCS*.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM TISSEC* (2009).

[5] Brad Antoniewicz. 2013. Analysis of a Malware ROP Chain. (Oct. 2013). http://blog.opensecurityresearch.com/2013/10/analysis-of-malware-rop-chain.html.

[6] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code. In *CCS*.

[7] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making Fine-grained Memory Randomization Practical by Allowing Code Sharing. In *USENIX Security*.

[8] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. of OSDI*. 423–436.

[9] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. 2003. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *USENIX Security*.

[10] Sandeep Bhatkar, R. Sekar, and Daniel DuVarney. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security*.

[11] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *CCS*.

[12] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *Security and Privacy*.

[13] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2016. Leakage-Resilient Layout Randomization for Mobile Devices. In *NDSS*.

[14] Kjell Bradeny, Stephen Crane, Lucas Davi, Michael Franz, and Per Larson. 2016. Leakage-Resilient Layout Randomization for Mobile Devices. In *NDSS*.

[15] Derek L. Bruening. 2004. *Efficient, transparent, and comprehensive runtime code manipulation.* Ph.D. Dissertation. MIT.

[16] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security 15*.

[17] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. 2017. NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE.

[18] Tzi-cker Chiueh and Fu-hau Hsu. 2001. RAD: a Compile-Time Solution to Buffer Overflow Attacks. In *ICDCS*.

[19] Mauro Conti and et. al. 2015. Losing Control: On the Effectiveness of Control-FlowIntegrity under Stack Attacks. In *CCS*.

[20] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Security and Privacy*.

[21] Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRAP: Table Randomization and Protection against Function Reuse Attacks. In *CCS*.

[22] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Canaries. In *ASIACCS*.

[23] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 555–566.

[24] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *NDSS*.

[25] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *USENIX Security*.

[26] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *ASIACCS*.

[27] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2013. Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM. In *ASIACCS*.

[28] Solar Designer. 1997. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/63. (1997).

[29] T. Durden. 2002. *Bypassing PaX ASLR protection.* Technical Report. Phrack Magazine, vol. 0x0b, no. 0x3b.

[30] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *Security and Privacy*.

[31] Bryan Ford and Russ Cox. 2008. Vx32: lightweight user-level sandboxing on the x86. In *USENIX ATC*.

[32] Jason Gionta, William Enck, and Peng Ning. 2015. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 325–336.

[33] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *Security and Privacy*.

[34] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. 2014. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *USENIX Security*.

[35] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack Davidson. 2012. ILR: where'd My Gadgets Go?. In *Security and Privacy*.

[36] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *ACSAC*.

[37] Vadim Kotov. 2014. Dissecting the newest IE10 0-day exploit (CVE-2014-0322). (Feb. 2014). http://labs.bromium.com/2014/02/25/dissecting-the-newest-ie10-0-day-exploit-cve-2014-0322/.

[38] Sebastian Krahmer. 2005. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://www.suse.de/~krahmer/no-nx.pdf. (2005).

[39] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *OSDI*.

[40] Lixin Li, James Just, and R. Sekar. 2006. Address-Space Randomization for Windows Systems.

[41] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*.

[42] the PaX team. 2001. Address Space Layout Randomization. http://pax.grsecurity.net/docs/aslr.txt. (2001).

[43] Matt Miller, Ken Johnson, Nitin Goel, and Vanegue Julien. 2011. Intra-modular Displacement Randomization. (2011).

[44] Ben Niu and Tan Gang. 2015. Per-Input Control-Flow Integrity. In *CCS*.

[45] Ben Niu and Gang Tan. 2014. Modular Control-Flow Integrity. In *PLDI*.

[46] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *CCS*.

[47] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking Holes in Information Hiding. In *USENIX Security*.

[48] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: defeating return-oriented programming through gadget-less binaries. In *ACSAC*.

[49] Vasilis Pappas, Michalis Polychronakis, and Angelos Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Security and Privacy*.

[50] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. Fine-Grained Control-Flow Integrity through Binary Hardening. In *DIMVA*.

[51] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *Proc. of EuroSys*. 420–436.

[52] Aravind Prakashm, Xunchao Hu, and Heng Ying. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *NDSS*.

[53] Manish Prasad and Tzi-cker Chiueh. 2003. A Binary Rewriting Defense against Stack Based Overflow attacks. In *USENIX ATC*.

[54] Rui Qiao, , and R. Sekar. 2017. Function Interface Analysis: A Principled Approach for Function Recognition in COTS Binaries. In *Dependable Systems and Networks*.

[55] Rui Qiao, Mingwei Zhang, and R Sekar. 2015. A Principled Approach for ROP Defense. In *ACSAC*.

[56] Jonathan Salwan. 2012. ROPGadget. http://shell-storm.org/project/ROPgadget. (2012).

[57] Felix Schuster, Thomas Tendyck, Liebcheny Christopher, Lucas Davi, Ahmad-Reza Sadeghiy, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Security and Privacy*.

[58] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: exploit hardening made easy. In *the 20th conference on USENIX Security Symposium*.

[59] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. Soffa. 2003. Retargetable and reconfigurable software dynamic translation. In *CGO*.

[60] Jeff Seibert, Hamed Okhravi, and Eric Söderström. 2014. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *CCS*.

[61] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS*.

[62] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-space Randomization. In *CCS*.

[63] Igor Skochinsky. 2012. Compiler Internals: Exceptions and RTTI. In *Recon*.

[64] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Security and Privacy*.

[65] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. 2015. Exploiting and Protecting Dynamic Code Generation. In *NDSS*.

[66] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *CCS*.

[67] PaX Team. 2002. *PaX SEGMEXEC*. Technical Report.

[68] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security*.

[69] Victor van der Veen, Dennis Andriesse, Enes Goktas, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *CCS*.

[70] Victor van der Veen, Enes Goktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *Security and Privacy*.

[71] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. In *SOSP*.

[72] Richard Wartell, Vishwath Mohan, Kevin Hamlen, and Zhiqiang Lin. 2012. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *CCS*.

[73] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Securing untrusted code via compiler-agnostic binary rewriting. In *ACSAC*.

[74] Jan Werner, George Baltas, Rob Dallara, Nathan Otternes, Kevin Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS)*.

[75] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[76] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2003. Transparent Runtime Randomization for Security. Florence, Italy.

[77] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity & Randomization for Binary Executables. In *Security and Privacy*.

[78] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. 2014. A Platform for Secure Static Binary Instrumentation. In *ACM Virtual Execution Environments*.

[79] Mingwei Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *USENIX Security*.

[80] Mingwei Zhang and R. Sekar. 2015. Control Flow and Code Integrity for COTS binaries: An Effective Defense Against Real-World ROP Attacks. In *ACSAC*.